Chapter 16: The make Utility

The UNIX **make** utility is a tool for organizing and facilitating the update of executables or other files which are built from one or more constituent files. Although **make** can be used in a wide variety of applications, in this chapter we concentrate on its use in the area of software development. We describe how to define relationships between source, object, library and executable files for use by **make**, and how to invoke **make** in its simplest and slightly more complex forms.

16.1 An Overview of the make Utility

make is a command execution utility. You can use it to essentially automate any task in which one or more "target" file(s) requires updating via a shell command when changes have been made to any of its "required" files (files from which the targets are built). There is some preparation to do, but once that is complete, all you do is enter the `make` command!

make compares the modification dates of target files to those of their required files. For each file that needs updating, **make** issues the predefined update command(s) for the file. For example, if file A is a required file of file B (the target), and if file A has a more recent "last modified date" than file B, then **make** *re-makes* file B by issuing the specified update command(s). Target files that are found to be more recent than all their required files are skipped over.

make is especially useful in long-term software development projects that involve large numbers of source files, libraries, and executables connected by a complex set of relationships. You can use it with any programming language whose compiler can be run with a shell command.

make obtains information about the files, their relationships, and the specific update commands from one or both of the following:

- a specially formatted, user-supplied control file called the *Makefile* (see section 16.2 *The Makefile and its Components*)
- **make**'s set of built-in default rules (see section 16.6 *make's Built-in Rules*)

In preparing to use **make**, you generally need to write a Makefile. The Makefile defines the relationships among the constituent and target files of your project by listing the required files for each target file, and stating the shell commands that must operate on the required files to create or update the target(s). **make** implicitly treats all required files as targets, in an iterative manner. A file listed as a required file in one definition statement may also explicitly appear as a target in another statement. For example, in a program, typically the executable file (the final target) is created from object files, which are in turn created by compiling source files. Once you have a working Makefile, you just run the **make** program to perform all your updating tasks.

The **make** man pages give a full description of the command, its options and features, as well as the format and usage of the Makefile.

16.2 The Makefile and its Components

The Makefile is a blueprint that you design and **make** uses to create or update one or more target files based on the most recent modify dates of the required files. The **`make`** command line syntax remains quite simple in the case where a Makefile is used:

```
% make [-f <makefile_name>] [<other options>] [<targets>]
```

If the **`-f`** option is not used, **make** checks for a Makefile of a particular name: it first looks in the current directory for a file named `makefile`, then for `Makefile`. If the Makefile is still not found, **make** looks in a couple of other places (see the man pages). In order to keep things simple and standard, we recommend that you always use the filename `Makefile` for your Makefile(s). Following this convention, you'll have only one Makefile in a directory. Your Makefile can contain instructions for building many different targets. When executing **make**, you can specify the desired target(s) on the **`make`** command line.

We can categorize the types of statements that can go in a Makefile as follows:

macro definition	A macro is a name that you define to represent a variable that may occur several times within the Makefile.
target definition	A target definition lists the target file, its required files, and the commands to execute on the required files in order to produce the target. (You can opt to specify the totality of this information in separate target definitions.)

suffix rules	Suffix rules indicate the relationship between target and source file suffixes (filename extensions). For example in FORTRAN, object files (`*.o`) are created from source files with a suffix of `f` (i.e. `*.f`). If no source file is explicitly given on a target definition line, **make** uses suffix rules to determine what source file to use to produce the target.
suffix declarations	Suffix declarations are lists of suffixes (file extensions) used in suffix rules.

Each new line in the Makefile starts a new definition, except that in target definitions:

- shell commands with leading tabs are part of the previous definition (a standard Makefile format that you need to recognize, but that we suggest you avoid using for reasons explained in section 16.2.2 *Targets*.)
- shell commands can be continued in standard UNIX format, using a trailing backslash (\)

Blank lines are permitted between definitions. Comments can be included after a pound sign (#). To use a literal pound sign, precede it with a backslash, i.e. \#.

16.2.1 Macros

A macro is a name that you define to represent a variable that may occur several times within the Makefile, or that needs to be updated frequently. Macros make maintenance of your Makefile much easier. They are commonly used to define settings, platform-specific commands, lists of required files for a target, lists of command options, and so on. **make** reads all the macro definitions before executing any commands. It is often convenient to put all the macro definitions at the head of the Makefile, but this is not necessary.

Format and Usage

Macro definitions are of the form:

<macro_name> = <value>

where **<macro_name>** is the name you want to use in place of the longer **<value>**. Everywhere in the Makefile that **<macro_name>** is found, **make** substitutes **<value>**. For portability, if **<value>** contains any blank spaces that it is supposed to have, the **<value>** string must be enclosed in quotes in the definition statement.[1] Backslashes can be used to continue the same line; you cannot put a new line in a macro value.

Once a macro is defined, you refer to its value in the form `$(<macro_name>)`. If **<macro_name>** is a single character, you can omit the parentheses.

As an example, let's define a macro FFLAGS to set some FORTRAN default options to use with the **f77** command:

```
FFLAGS = "-O2  -w -Olimit 1500 -nocpp "
```

You can now refer to the value of FFLAGS within the Makefile in the form `$(FFLAGS)`. For example, you might include a target definition line like the following (the format is explained below in section 16.2.2 *Targets*):

```
foo : foo.f ; f77 -o foo $(FFLAGS) foo.f
```

Special Macros

To be sure that **make** uses the standard, portable Bourne shell, always include in your Makefile a macro of the form:

```
SHELL = /bin/sh
```

Some versions of **make** default to your current interactive shell if you don't include this explicit SHELL macro in your Makefile. The standard Makefile format that we describe in this chapter assumes **sh** as the command interpreter.

Macro Sources

Macro definitions are similar to and can take default values from environment variables. **make** gets additional macro definitions from the following sources, and applies them in the order shown:

1) currently defined environment variables

---

1. Be aware that this quoted value is what **make** hands to the shell. The shell then hands the macro, <e.g., `$(FFLAGS)` to a program to execute. If you want leading or trailing spaces in `<value>` to be included in the command, specify the macro in double quotes in the command statement, e.g., `"$(FFLAGS)"`.

2) built-in **make** rules

3) definitions in the Makefile

4) definitions on the command line

In other words, this list is in order of reverse precedence; a value from a source later in the list overrides a value applied from an earlier one. Using the **`-e`** option on the **`make`** command line swaps the first two; see section 16.3.1 *General Usage*.

Symbols Used in Macros

`$$`	maps to a literal dollar sign
`$*`	is used in suffix rules (see section 16.2.3 *Suffix Rules*); it refers to the filename without the suffix
`$@`	is the current target make is processing
`$<`	is the implied source in a suffix rule

16.2.2 Targets

Targets are the files that you want to update or create. A complete target definition includes the name of the target, the files needed to build it (its required files), and the commands that must be executed to recreate the target.

Format

The standard Makefile format using Bourne shell conventions calls for:

- a space between listed targets
- a colon (**`:`**) between the last target and the first required file
- a space between listed required files
- a semi-colon (**`;`**) after the last required file if commands follow on the same line
- a semi-colon (**`;`**) between the successive listed commands

A simple target definition with a single target, required file, and command can be written in the form:

```
<target> : <required_file> ; <shell_command>
```

or, using a more traditional format, listing the command on the next line (note that the semicolon (**`;`**) is omitted here):

```
<target> : <required_file>
{tab}<shell_command>
```

In this traditional Makefile format, the commands beyond the first line of a definition must have leading tabs, and there must be no intervening blank lines.

There are two reasons to avoid this second format: first, when working with the Makefile it is hard to see the difference between a tab and blanks, and secondly, some editors change tabs to blanks (or vice versa), which causes problems (neither **emacs** in default mode nor **vi** changes tabs to blanks). You can check your file to see if it contains tabs or blanks by running the command:

```
cat -tev <filename>
```

We propose as an alternative that you use the backslash character (****) to continue the single definition line down as many physical lines as you have to go. Also, note that under AIX the standard "tabbed" format can be unpredictable; it is therefore safer for *several* reasons to use our suggested format on this platform.

This "safer" format which we suggest if the entire definition doesn't fit on the first line is:

```
<target> : <required_file>; \
<shell_command>
```

A target definition line can contain more than one of each element type. Or, it may contain only two of the three element types. A more complex definition in our suggested format looks like:

```
<target_1>    <target_2>    ...    :    <required_file_1>
<required_file_2> ... <required_file_n> \
<required_file_n+1> ...; \
<shell_command_1> ; <shell_command_2> ; <shell_command_3>
; ... ; \
.
.
.
... ; <shell_command_m>
```

If there is more than one target (e.g. `<target_1> <target_2>`) in one definition, the commands are attempted separately for each target.

If you find it easier, you can list multiple required files for a single target in separate target statements. However only one statement for a given target can include commands, and therefore must include all the commands. Here is an example of this alternative format:

```
<target> : <required_file_1>
<target> : <required_file_2>
<target> : <required_file_3>
<target>  :  ;  <shell_command_1>  ;  <shell_command_2>  ;
<shell_command_3> ; ...
```

It can be confusing if you separate a series of statements like this from one another in the Makefile; if you use this format, keep the statements together!

Usage

The relative modification times of the `<target>` and the `<required_file>`(s) determine whether the listed commands will be executed. If the target file is found to be missing or to be older than any of its required files, **make** executes the commands to rebuild it. **make** treats the required files iteratively as targets, whether or not they are explicitly listed as targets in subsequent target definitions, and rebuilds them as necessary before rebuilding the final target.

16.2.3 Suffix Rules

A suffix is essentially a file extension. A suffix rule defines the relationship between target and required files by their file extensions. A suffix rule is much like a target definition except that it uses implied rather than explicit file names for target (output) and required (input) files.

A suffix rule is of the form:

```
.<insuffix>.<outsuffix> : ; <command>
```

The dots are really part of the suffixes themselves. As an example, assume you have a set of target files to rebuild whose filenames are all of the form `*.b`. The required files for these targets have filenames of the form `*.a`. Define the suffix rule:

```
.a.b : ; <command(s)>
```

make expands this to the following target definition for all files ending in `.a` in the current directory:

```
<filename>.b : <filename>.a ; <command>(s)
```

If you define suffix rules specifically for intermediate files in a process, you still need to include a rule for the final and original files, for example, if you define:

```
.tex.dvi: ; latex $*  #latex cmd makes .dvi files
from .tex files
.dvi.ps: ; dvips $*   #dvips cmd makes .ps files from
.dvi files
```

You still need to provide the rule:

```
.tex.ps: ; latex $*; dvips $*  #make .ps files from
.tex files
```

make is not sophisticated enough to do the transitive closure on suffix rules.

Note that suffixes don't have to start with a dot (`.`).

16.2.4 Suffix Declarations

Any suffix that you use in a suffix rule must be listed explicitly in a `.SUFFIXES` declaration in the same Makefile unless it is included in **make**'s built-in suffix declarations (see section 16.6 *make's Built-in Rules*). A suffix included in the built-ins can also be included in a suffix declaration in the Makefile.

Suffix declarations are really target definitions for a special target named `.SUFFIXES` and they contain no commands.[1] They are of the format:

```
.SUFFIXES : .a .b
```

where `.a` and `.b` are suffixes. This example suffix declaration would allow you to include the suffix rule from 16.2.3 *Suffix Rules* in your Makefile:

```
.a.b : ; <command(s)>
```

You may combine all the suffixes you use in the Makefile into one `.SUFFIXES` declaration, or group them into separate statements.

Suffixes that you declare add to the built-ins; they do not replace them.

16.2.5 Control Files within a Makefile

You may encounter situations where it is useful to pipe input or output of one command to another within a Makefile. You can echo a small control or data file within the Makefile, rather than maintaining a separate external file. In the following target definition example from an Isajet Makefile, several control statements are echoed into the **patchy** utility in order to extract `isaint.f` from the `isajet.car` patchy library:

```
isaint.f : isajet.car ; \
(echo "+USE,*ISAJET,$(MACHINE).              "; \
 echo "+USE,INTERACT.     INTERACTIVE PATCH  "; \
 echo "+EXE.                                 "; \
 echo "+PAM,T=C.                             "; \
 echo "+QUIT.                                ";)\
| ypatchy isajet.car  isaint.f  \& genint.lis .GO
```

1. Most versions of **make** have other special targets (sometimes called magic targets) besides `.SUFFIXES`. These special target names always start with a dot (`.`).

16.3 Running make

16.3.1 General Usage

The **make** utility is invoked with the **`make`** command. The command syntax is:

```
% make [<options>] [<targets>]
```

Several **`<options>`** are available, and are described in the man pages for **make**. We provide a list of some of the commonly used options:

`-n`	Preview the commands, don't execute. Very useful for testing.
`-d`	Debug; list the operations used and why ("read **make**'s mind"). Available on all supported platforms.
`-e`	Environment variables override built-ins.
`-f <makefile_name>`	If your Makefile has a name other than `makefile` or `Makefile`, use this option followed by the file's name to identify it (leave a space between the option and the filename).
`-p -f /dev/null [|less]`	Print the built-in rules (see section 16.6 *make's Built-in Rules*).

The **`<targets>`**, as mentioned earlier, are the files that you want to create or update. **make** searches the Makefile to find a target definition statement for each target listed on the command line.

You can include macro definitions on the command line which get applied after the assignments made in the Makefile. For example:

```
% make "CC = gcc" <target>
```

16.3.2 Usage without Specifying Target

When **make** is invoked without a specified target, the first non-suffix target in the Makefile is used. The command is simply:

```
% make [<options>]
```

For larger products, it is standard practice to name this target *all* in the Makefile, and in the list of required files to list the individual targets which together actually produce the full product. For example, the first Isajet target definition is:

```
all : isadecay.dat \
```

```
        isatext.doc  \
        isajet.a     \
        isaint       \
        isasusy
```

Notice that here no commands are listed. They would appear in the subsequent target definitions.

16.3.3 Usage without a Makefile

The extensive built-in rules let you use **make** quite effectively without having your own Makefile. Section 16.6 *make's Built-in Rules* provides a brief explanation of the built-in rules. **make** will look for any file whose name matches that specified on the command line and which has a file extension that identifies it as a reasonable source. For example, to produce the executable `foo` (the target) from a `foo.c` or `foo.f` source that exists in the current directory, enter:

```
% make foo
```

make will look for the C or FORTRAN file as the source file for this target. Taking the FORTRAN program and no options as an example, this command is equivalent to (see section 16.2.3 *Suffix Rules*):

```
% f77 -o foo  foo.f
```

16.4 "Housekeeping" Targets

It is common practice to have a "housekeeping" target which removes stray files from the working directories. Typically you would run **make** on this target after you've completed the **make** operation on your principle target(s). It is conventional to call this target *clean*. Here is an example which removes unnecessary generated files from several different directories. The target definition has no required files:

```
clean: ; \
  rm  -f *.bak    ;\
  rm  -f *.lis    ;\
  rm -rf Maketemp ;\
  cd example/isaplt ; rm -f *.lis*   ;\
  cd  ../jet        ; rm -f jet.log*
```

You need to determine what stray files will be generated in your case, and define your commands accordingly. Run **make** on the *clean* target by entering:

```
% make clean
```

You may wish to define different levels of housekeeping targets. One that clears out everything, leaving only the original files you had before running **make**, is often named *clobber*.

16.5 Portability

It is desirable for your Makefile to be portable across different UNIX platforms. Why might this be a problem to implement? As mentioned earlier, **make** does all macro processing before any commands are executed. Therefore environment variables set in shell scripts executed by **make** have no effect on **make**'s macro definitions. And standard **make** doesn't support conditional macro definitions. So, how can you write a portable Makefile?

A Solution

1) Create a script for setting environment variables.

2) Have **make** run this script ("source" it; see section 5.4 *Shell Scripts*) from within the Makefile.

3) After it runs the script, have **make** rerun itself with a different target and the original command line options. A *$(MAKE)* macro which causes **make** to rerun itself is a standard feature.

An example of this technique follows.

Example

Create a portable shell script (named, for example, `Makeenv`) which sets appropriate environment variables for the Makefile. Here is a simple `Makeenv` script which defines the macro `F77` based on the current platform as determined by the command **`uname -s`**:

```
export F77 MACHINE

case `uname -s`  in
IRIX)
        F77="f77  -O2  -w -Olimit 1500 -nocpp "
;;
OSF1)
        F77="f77  -O1  -w -Olimit 1500 -nocpp -static "
;;
esac
```

The Makefile is below. Run **make** with the target `isaint`. The Makefile runs `Makeenv`, then **make** reruns itself with the target `do_isaint` and the correct `F77` value:

```
isaint    : isaint.f ;  . Makeenv; $(MAKE) do_isaint
do_isaint : isaint.f ; $(F77) isaint.f  -o isaint
```

Other Utilities

There are other utilities available for more complicated cases:

- **gmake (**Gnu make**)**, part of the Fermilab **gtools** product, has some nice portability features, and supports other advanced features like parallel compilation on multiprocessor systems.
- There are preprocessors for building locally tailored Makefiles in very sophisticated ways, including **gnu configure**, **premake**, and **imake**.

16.6 make's Built-in Rules

make comes equipped with a long list of built-in defaults to make your job easier. You are free to override any of them in your Makefile. The defaults fall roughly into four categories:

1) macros that reflect your current environment variables

2) macros that define standard compilers and options

3) suffix rules for finding required files when building targets

4) a list of known suffixes

To get a listing of all the built-in macros and rules, enter the command:

```
% make -p -f /dev/null [| less]
```

Depending on your platform, there may be nearly a thousand lines of definitions, so you might want to pipe this to **less**, or redirect the output to a file.

16.7 A Few Caveats...

1) Recall that in the traditional Makefile target definition format successive commands are entered on successive lines, each starting with a tab. Be aware that each of these command lines runs in a different shell. Two important implications of this are:

   a) if you have issued a change directory (**`cd`**) command, it does not carry over to the following line(s)

   b) environment and shell variables do not carry over to the following line(s). (The format which uses a single logical line for the entire definition avoids this problem.)

2) If you use non-shell commands (for example **`ls`**) in definition statements, be aware that the output may vary from platform to platform. For this reason it is best not to rely on the specific output format of these commands.